# Kinesthetics eXtreme: An External Infrastructure for Monitoring Distributed Legacy Systems

Gail Kaiser[1], Janak Parekh[1], Philip Gross[1] and Giuseppe Valetto[1,2]
[1]*Columbia University Programming Systems Lab, Department of Computer Science*
[2]*Telecom Italia Lab*
*{ kaiser, janak, phil, valetto } @ cs.columbia.edu*

## Abstract

*Autonomic computing – self-configuring, self-healing, self-optimizing applications, systems and networks – is widely believed to be a promising solution to ever-increasing system complexity and the spiraling costs of human system management as systems scale to global proportions. Most results to date, however, suggest ways to architect* new *software constructed from the ground up as autonomic systems, whereas in the real world organizations continue to use stovepipe legacy systems and/or build "systems of systems" that draw from a gamut of new and legacy components involving disparate technologies from numerous vendors. Our goal is to retrofit autonomic computing onto such systems,* externally*, without any need to understand or modify the code, and in many cases even when it is impossible to recompile. We present a meta-architecture implemented as active middleware infrastructure to explicitly add autonomic services via an attached feedback loop that provides continual monitoring and, as needed, reconfiguration and/or repair. Our lightweight design and separation of concerns enables easy adoption of individual components, as well as the full infrastructure, for use with a large variety of legacy, new systems, and systems of systems. We summarize several experiments spanning multiple domains.*

## 1. Introduction

The increasing complexity of computer systems, networks and applications has led to a tremendous interest in what some have termed *autonomic computing* [1]: in particular, the notion of self-managing software is an attractive approach to reducing the time and effort costs of maintaining and operating software systems. Such technologies are now being promoted in the COTS market; for example, Microsoft's XP product line has debug and repair semantics built-in to try and reduce downtime [2]. However, such approaches often ignore legacy software, assuming the user will be willing and able to migrate. A New York Times article [3] describes the "trailing edge" industry, where migration usually is not an option. The article is primarily concerned with electronic components and other hardware used by the military, but the author notes similar factors are at play in civilian telecommunications equipment, medical devices, etc. – many of which also run old software. Even when not running on archaic hardware, legacy software may be implemented in "unsafe" languages like C, or written in languages no longer in common use, making the need for autonomic repair even greater [4].

Various facilities have been developed to automate problem detection and/or repair. For example, some new operating systems and applications include engines to automate the collection of crash data [5]; other tools help detect anomalous behavior by monitoring system and application logs [6]; and a few tools provide administrative control over application behavior [7]. However, these tools are largely application-neutral, leaving understanding of what the system is supposed to be doing, how and why, to the human administrator. Thus only the simplest general-purpose analyses and repairs can be automated.

In an attempt to do better, we have developed a generic framework for collecting and interpreting application-specific behavioral data at runtime, tailored to the application (or more generally to the domain) by introduction of models that describe both expected correct behaviors as well as anticipated error situations. What we call *probes* are attached to the target system to collect data, while *gauges* aggregate, filter and interpret the probed data. This monitoring framework can be used with or without a feedback loop that automatically performs repairs and reconfigurations; we present our automated repair framework separately in [8].

Our implementation of this approach is called Kinesthetics eXtreme, or KX. KX runs as a lightweight, decentralized, easily integrable collection of active middleware components, tied together via a publish-subscribe (content-based messaging) event system. We show how KX can be used to monitor a variety of target applications employing application-level semantics. XML is used as a native data format, providing rich expressiveness.

## 2. Background

# Report Documentation Page

*Form Approved*
*OMB No. 0704-0188*



| 1. REPORT DATE | 2. REPORT TYPE | 3. DATES COVERED |
|---|---|---|
| **2005** | | **-** |

| 4. TITLE AND SUBTITLE | 5a. CONTRACT NUMBER |
|---|---|
| **Kinessthetics eXtreme: An External Infrastructure for Monitoring Distributed Legacy Systems** | 5b. GRANT NUMBER |
| | 5c. PROGRAM ELEMENT NUMBER |
| 6. AUTHOR(S) | 5d. PROJECT NUMBER |
| | 5e. TASK NUMBER |
| | 5f. WORK UNIT NUMBER |
| 7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Defense Advanced Research Projects Agency,3701 North Fairfax Drive,Arlington,VA,22203-1714** | 8. PERFORMING ORGANIZATION REPORT NUMBER |
| 9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) | 10. SPONSOR/MONITOR'S ACRONYM(S) |
| | 11. SPONSOR/MONITOR'S REPORT NUMBER(S) |

12. DISTRIBUTION/AVAILABILITY STATEMENT
**Approved for public release; distribution unlimited**

13. SUPPLEMENTARY NOTES
**The original document contains color images.**

14. ABSTRACT
**Autonomic computing - self-configuring, self-healing, self-optimizing applications, systems and networks - is widely believed to be a promising solution to ever-increasing system complexity and the spiraling costs of human system management as systems scale to global proportions. Most results to date, however, suggest ways to architect new software constructed from the ground up as autonomic systems, whereas in the real world organizations continue to use stovepipe legacy systems and/or build "systems of systems" that draw from a gamut of new and legacy components involving disparate technologies from numerous vendors. Our goal is to retrofit autonomic computing onto such systems, externally, without any need to understand or modify the code, and in many cases even when it is impossible to recompile. We present a meta-architecture implemented as active middleware infrastructure to explicitly add autonomic services via an attached feedback loop that provides continual monitoring and, as needed, reconfiguration and/or repair. Our lightweight design and separation of concerns enables easy adoption of individual components, as well as the full infrastructure, for use with a large variety of legacy, new systems, and systems of systems. We summarize several experiments spanning multiple domains.**

15. SUBJECT TERMS

| 16. SECURITY CLASSIFICATION OF: | | | 17. LIMITATION OF ABSTRACT | 18. NUMBER OF PAGES | 19a. NAME OF RESPONSIBLE PERSON |
|---|---|---|---|---|---|
| a. REPORT | b. ABSTRACT | c. THIS PAGE | | | |
| **unclassified** | **unclassified** | **unclassified** | | **9** | |

The monitoring model, as inspired by the DARPA DASADA program [9], is explicitly designed to be lightweight. There are several first-class entities:

- The *target system* refers to the application, set of applications, or application components that are being monitored;
- *Probes* are generally small, constrained, noninvasive pieces of code which get installed in or around the target application system – they may inject source code, modify bytecodes or binaries, replace DLLs or other dynamic libraries, inspect network traffic, and/or perform other related tasks to collect this information;
- *Gauges* are responsible for interpreting data from these probes, and generate *semantic* events about the behavior of the application - often operating in an effective hierarchy where higher-level gauges interpret aggregate events from lower-level gauges;
- *Controllers* receive analysis results from the gauges, and decide if and when to coordinate one or more effectors to attempt a repair.
- *Effectors* apply reconfiguration or repair, usually tuning or replacing an individual component, or spinning up a new component, as per the task(s) defined by relevant controllers.

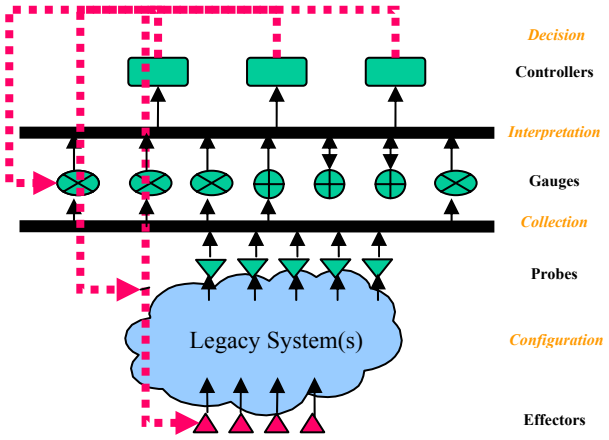


**Figure 1. Overview of the monitoring infrastructure.**

Figure 1 shows the data (thin solid lines) and control flow (thick dashed lines) of the monitoring infrastructure on top of a target system. Note that this is a conceptual diagram; the buses may be unified, separate, or there may be no bus per se, instead, point-to-point connections could be used (although the intent is efficient multicast based on content subscriptions). We take advantage of this to loosely couple the probes, gauges and controllers in our architecture by making all of them *event-based*, i.e., every component asynchronously messages each other via a standardized event middleware. We currently use U. Colorado's *Siena* publish-subscribe system [10], which supports events represented as collections of attribute-value pairs. We are also actively developing our own event system, Multiply Extensible Event Transport (*MEET)*, to natively support richer event formats, such as XML, as well as performance optimizations. By leveraging such event middleware, KX components can be easily rearranged, or multiple instances of KX components can be used, to address the needs and scalability requirements of the target system.

## 3. Implementation

Our KX implementation is composed of probe, gauge, controller and effector components in order to accomplish system monitoring and reconfiguration.

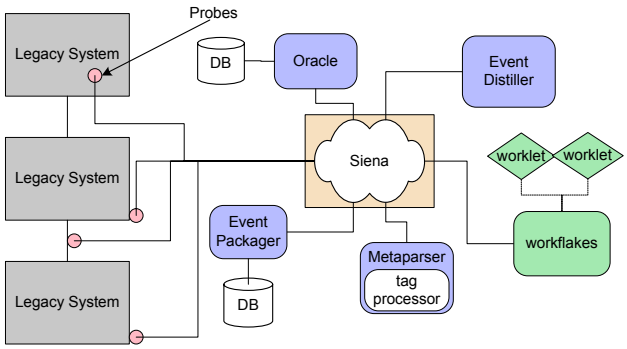


**Figure 2. KX System Overview.**

A specific probe technology is not formally part of the KX infrastructure, as the best selection among potential technologies is often peculiar to the implementation details of the target system and can vary widely. We have employed a number of probe solutions developed by others, which we discuss in a later section (see section 4, Experiments).

The *Event Packager* provides event translation and "flight recorder" services to standardize and log all of the incoming event streams from the probes. The *Event Distiller* performs sophisticated cross-stream temporal event pattern analysis and correlation to monitor desirable (and, correspondingly, undesirable) behavior. Both of these components are discussed in detail in the next few sections, and constitutes the main contribution of this paper.

The *Metaparser* and *Oracle* are optional components that can be utilized to perform intelligent XML-based event vocabulary discovery and ontology translation, based on our separate Flexible XML (FleXML) effort. They are most useful when KX is used for target systems and/or probe technologies that natively output XML-

formatted events. The Metaparser and Oracle are discussed in [11].

Finally, *Workflakes* and *Worklets* are our control and effector technologies, respectively. Workflakes combines a decentralized workflow engine based on the open-source Cougaar technology [12] with Worklets, which is our own mobile agent architecture [13]. A complete discussion of Workflakes and corresponding control and repair scenarios is described in detail in [8].

It is important to note that all of these are separately usable components. Depending on the problem domain, one or more of these components may be used. For example, if only a few very well-defined repair scenarios are to be performed, or if KX is only being used to do high-level monitoring without reconfiguration or repair, one may choose to omit the Workflakes component. Similarly, if only one source of events is being monitored, the Event Packager component may be redundant. In fact, the development of Worklets preceded the rest of KX, and was originally developed for other purposes [14], but adapted nicely to our reconfiguration and repair requirements. Siena will soon replaced by our own MEET, but other event propagation technology could be easily dropped in as elaborated below.

Freely-available downloads of all KX components, as well as the full system, may be obtained at [15].

### 3.1 Event Packager

The Event Packager component is designed to support event aggregation, transformation, and persistent spooling. It utilizes a plug-in architecture to support a variety of incoming event formats (*inputs*), including XML, Siena, Elvin [16], SMTP, raw TCP data, Java RMI, etc.; a variety of transformations, including the persistent spooling and timestamping; and a variety of output options, closely mirroring the input possibilities. New plugins can easily be created; for example, we are working on integrating Instant Messaging (IM) protocols to support a richer variety of event systems.

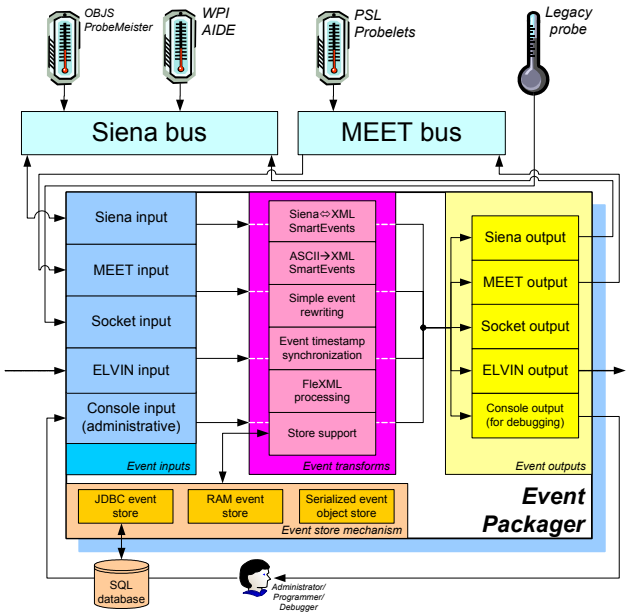


**Figure 3. Event Packager.**

The various plugins are coordinated via a user-definable XML rulebase that dictates what should be done to the data (*transforms)* and to whom the data should be sent. Typically, a number of different input formats are streamlined, spooled, and aggregated onto one event stream for the other KX components. Appendix A lists an example Event Packager rulebase. The Event Packager also supports dynamic rulebase addition by specially-formed events sent to it.

Internally, the datapaths as defined by the rulebase are implemented as fast pipelines, so that events introduced into the Event Packager are routed very quickly to the appropriate transforms and outputs. In particular, incoming events are wrapped in a format-neutral Event Packager event, and are tagged so the Event Packager can route them amongst transforms and outputs with only minimal inspection.

The Event Packager is currently about 9,000 lines of Java code; the core engine is about 2,000 lines, while bundled plugins (including, but not limited to, the aforementioned) is the rest; there is also some small amount of C glue code to handle sendmail and other legacy integration.

### 3.2 Event Distiller

In many monitored systems, the key is to determine what original failure ("root cause") started a cascading problem [17]. The Event Distiller is the component responsible for detecting causality amongst the events in significant event sequences, by performing time-based pattern matching. Internally, it uses a collection of nondeterministic state engines for temporal complex

event pattern matching. While this is memory-intensive, it allows a richer representation of event sequences: logic constructs are supported, as are loops, rule chaining, and variable binding. We also mitigate memory usage by supporting timeouts and automatic garbage collection. Timestamped event reordering is also supported, so if events arrive out-of-order within a certain window, the Event Distiller will rearrange them appropriately so that sequences, and causality, can still be recognized correctly.

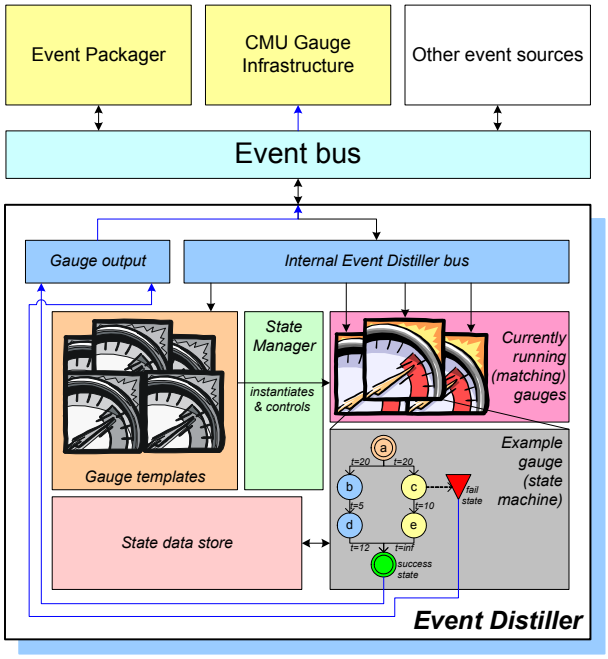


**Figure 4. Event Distiller.**

Event Distiller rules may be populated in one of several ways: First, an XML rulebase (separate from the Event Packager rulebase) is supported, where event sequences are specified, along with timebound parameters as well as "success" and "failure" notifications; we have also developed a GUI to assist a KX integrator; it also works as a systems management console for human engineers, although our goal is to automate many repairs within a KX feedback loop. Second, the Event Distiller supports dynamic rule generation – messages can be sent to the Event Distiller with XML snippets specifying a rule or a segment of a rule (e.g., to construct new rules on the fly or modify existing rules). Third, as with the Event Packager, other sources can be easily integrated: We have integrated support for ACME [18] constraints – the Event Distiller can act as a "reporting gauge" onto CMU's ACME Gauge Bus, thereby providing feedback to the architectural description language and corresponding architecturally-oriented repair tools. We are also investigating learning techniques to build rules in a more autonomic fashion.

Currently, the Event Distiller has two different execution models: it can either exist as a Siena client or can be embedded directly inside another KX (or target system) component, at which point events are passed through method calls. If XML event formatting is involved, the Metaparser can be situated in front of the Event Distiller to perform reduction; we are currently developing the next version of the Event Distiller, which will handle fast XPath matching directly.

The Event Distiller is implemented in Java and is currently about 7,000 lines of code; a typical rulebase is usually a few hundred lines of XML

## 4. Experiments

In order to validate KX's utility as a monitoring infrastructure, we developed several scenarios and corresponding experiments with real, deployed complex distributed software. We describe three scenarios in this section: *email processing*, *failure detection*, and *load balancing*.

### 4.1 Email processing (spam detection)

Email spam has become a persistent nuisance [19]. Often, an identical message gets sent to a huge number of targets (e.g., all addresses from a virus-recipient's address book or web-spider-constructed mailing list).

In order to demonstrate KX's flexibility beyond conventional failure detection, reported in previous papers, we instrumented Sendmail [20], a popular email Message Transfer Agent (MTA), to capture messages being received in a target network. More precisely, a *Sendmail milter* [21] was installed to capture incoming traffic to the Event Packager. Specific attributes about each message (such as source address, subject, or Message-ID) were captured by probes, encapsulated into events, and sent through the Event Distiller. The Event Distiller was fed with rules that would trigger if multiple (3+) messages containing the same source and Message-ID were received in a very short timespan (less than 10 seconds).

```
<state name="a" timebound="-1" children="b">
  <attribute name="from" value="*1"/>
  <attribute name="subject" value="*2"/>
</state>
<state name="b" timebound="100" count="1" children=""
       actions="A,B" fail_actions="F" absorb="true">
  <attribute name="from" value="*1"/>
  <attribute name="subject" value="*2"/>
</state>
```

**Figure 5. Sample pattern to detect repeated emails.**

A number of different constructs are used here to allow for flexibility. The "timebound" construct was previously alluded to; note that an initial event in a pattern implicitly does not have a timebound. "Children" designates successors from a given state, and "Actions" and "Fail_Actions" denote success and failure, respectively, and refer to notification specified elsewhere in the rulebase. (A full example, including notifications, is in the appendix; comprehensive documentation on the rule language can be found at [22].)

Additionally, the "*1" term in the above rules designate a *wildcard binding*, i.e., the Event Distiller substitutes all instances of "*1" by the first source that it sees for this instance of the rule. By doing so, the Event Distiller is able to leverage one rule to match a large number of different sources and subjects.

Once detection has occurred, resolution is accomplished by dispatching a Worklet that reconfigures the Sendmail MTA in the target network to block all further messages from that source address by rewriting the configuration file and sending a hangup signal (SIGHUP) to Sendmail to reload its configuration. In our experiments, the solution worked for simple spam – i.e., one message sent by a spammer to a broad number of people in the same organization would verifiably get caught and future communication from that spammer would be blocked.

While this technique has been superceded by better spam-specific technologies, such as SpamAssassin [23], which uses dynamic rules and Bayesian learning to distinguish more "stealthy" spam, this example demonstrates the broad utility of our Event Distiller's timebound-based pattern matching, even with email-specific semantics. In essence, we were able to add (limited) autonomic behavior to Sendmail.

## 4.2 Failure detection

We also integrated the KX infrastructure with a complex GIS mapping system developed at ISI and used experimentally at PACOM, known as GeoWorlds [24]. GeoWorlds is built out of a distributed set of services glued together by Jini [25]. While the system generally works well, there are very complex services that occasionally stop running correctly, and there's normally no recourse except to wait for the request to time out and to manually restart the appropriate backend component.

Using WPI's AIDE [26] probe technology, we were able to automatically instrument the GeoWorlds Java source code with probes that would monitor the start and end of method calls that were relevant to this service. The Event Distiller then incorporated rules to monitor a variety of method calls, making sure that a "termination" call matched up with each "initiation" call within an appropriate timebound (ranging from seconds to a minute). AIDE reported method calls in an XML format; these calls were then be translated to a simple attribute-value set via the Metaparser and fed into the Event Distiller. The following XML is an example of the incoming event patterns used to perform such failure detection.

```
<state name="Start" timebound="-1" children="End" actions=""
        fail_actions="">
  <attribute name="Service" value="*service"/>
  <attribute name="Status" value="Started"/>
  <attribute name="ipAddr" value="*ipaddr"/>
  <attribute name="ipPort" value="*ipport"/>
  <attribute name="time" value="*time"/>
</state>
<state name="End" timebound="15000" children=""
        actions="Debug" fail_actions="Crash">
  <attribute name="Service" value="*service"/>
  <attribute name="State" value="FINISHED_STATE"/>
  <attribute name="ipAddr" value="*ipaddr"/>
  <attribute name="ipPort" value="*ipport"/>
  <attribute name="time" value="*time2"/>
</state>
```

**Figure 6. Failure detection pattern.**

In particular, the incoming probes reported Status and State values that were closely watched to track method completion. If for some reason a "FINISHED_STATE" was not received within 15 seconds after a method had initiated, the system would send out the "Crash" event; otherwise, the "Debug" notification would be sent out. (Both notifications can be seen in a larger example in the Appendix.)

In this case, if the repair system received a "Crash" event, the repair involved would be a simple restart of the service. A more sophisticated repair could coordinate multiple services to prevent having to restart the operation that triggered the crash in the first place. Even in the first case, however, we were able to automate a process that, previously, had been done manually.

## 4.3 Load balancing/QoS

**4.3.1 Load-balancing GeoWorlds.** In addition to developing failure detection for GeoWorlds, we implemented a load-balancing solution, as a number of different GeoWorlds execution scripts rely on computationally-intensive backend services; crash avoidance and performance maximization through request relocation was clearly desirable. To accomplish this, we utilized the relocatibility of Jini services to build a load-balancing solution for GeoWorlds. First, a system monitor probe was built in C# to measure the overall load on the system, and results were piped into a custom plugin for the Event Packager. Second, an ACME

architectural description of the GeoWorlds system was created, which included specified load constraints on the appropriate services. The rules were then dynamically generated and fed into the Event Distiller based on the predefined architectural constraints.

During the execution of various services, if this load would exceed a predetermined threshold (defined as a constraint in an ACME architectural description of the GeoWorlds system) for an extended period of time, the Event Distiller would detect and report it as a violation of the architectural constraints – and the triggered repair would cause the service to move to a different Jini-enabled host. Additional logic was programmed into the Event Distiller rulebase to detect oscillation (thrashing) and proactively prevent it. We were also able to visualize the load and service state using AcmeStudio's [27] architectural diagram visualization tools.

**4.3.2 TILAB IM System.** TILAB [28] has developed and deployed a J2EE-based multi-channel Instant Messaging (IM) service, which is currently used daily by thousands of end-users [8]. KX was validated in this scenario for autonomically handling a variety of monitoring, reconfiguration and repair requirements of the service architecture. First, on-demand scalability is supported: by probing user sign-on events and server request queues, KX can determine the load of each element in the IM server farm and take appropriate actions whenever needed [8]. Repairs, selected on the basis of the inferences carried out using Event Distiller rules, encompass modifications to the threading model of active servers or on-the-fly deployment and activation of additional server instances and corresponding reconfiguration of the commercial load-balancer (an IBM product in this real-world configuration) to redirect client traffic to these new servers. Failure detection is also supported from a load-balancing standpoint: information on server failures, as well as interconnections between servers and the backend DBMS entities is similarly captured to facilitate load balancer reconfiguration to direct client traffic to still-functional servers. The same set of probes and actuators, coupled with slightly different Event Distiller gauge rules and Workflakes repairs, can also be used to support controlled and graceful staging of the service infrastructure; this enables automated software release deployment without necessitating a complete shutdown (and service interruption) during the transition.

Overall, the TILAB case study demonstrated benefits with respect to application-level QoS, as well as ease and automation of service management. More generally, it proves KX's utility and applicability in an industrial context.

Most of the above scenarios were implemented with minimal manually-written glue code for attaching our external autonomic infrastructure to the target system. In the GeoWorlds case, we were able to utilize a pre-existing tool for automatic instrumentation; if source was not available, tools like OBJS' Software Surveyor [29], which is capable of runtime Java bytecode munging, could be used instead. The KX overlay is extremely lightweight, and by configuring application-specific rules, we've been able to add a variety of system monitoring-related functionality that was not originally embedded in nor planned for the target system. If other monitoring is desired, minor rule reconfiguration can be performed during runtime without shutting down either the target system or the KX monitoring infrastructure.

## 5. Related Work

Other projects in the DARPA DASADA program more directly addressed the technical details of system assembly, adaptation, and reconfiguration; one of our main goals was to provide standardized infrastructure to support their gauges and extend their repairs to real-time processing, while the target system remained running (without bringing it "down"). Garlan [30] discusses static model checking, Geib [31] performs formal verification, and Osterweil [32] and Wolf [33] address reconfiguration workflow. In contrast, our gauges are more focused on dynamic application monitoring.

The Astrolabe project [34] uses a replicated DNS-like infrastructure to support a number of applications, including system monitoring semantics; although we also consider Internet-scale applications, their approach may be better suited to a more distributed monitoring model, where many nodes need to know the information and it is acceptable for latencies to be in the tens of seconds.

The NESTOR project [35] takes a network-layer approach to monitoring. In the commercial arena, OC Systems has an analogous platform to DASADA probes and monitors with their AProbe [36] and RootCause [37] products, while SMARTS offers their Automated Business Assurance service with "Codebook Correlation Technology." [38] These technologies are generally noninvasive and rely on quickly matching against static or predetermined analysis, as compared to our intent to integrate with application semantics, where new success or failure rules can be introduced on the fly.

Fault management systems [39,40] are also closely-integrated at the systems level, for telecommunications-level reliability. These systems are largely static, designed for vertical solutions, and not for complex distributed "systems of systems".

Intrusion detection systems [41,42] usually focus on system- or network-level security, and are not generally

useful for application reliability or self-management. We are actively investigating the migration of our work towards intrusion detection to better support specific application-level security semantics, in particular based on semantic models gleaned from machine learning systems [43].

A number of academic and commercial generalized event correlation systems exist, which correspond, to some extent, to our Event Distiller gauges [44,45]. These generally use a coding and compilation approach to defining event patterns; in contrast, our dynamic-at-runtime rules are better adapted to embedding solutions in continuously running systems, albeit at potentially lower performance levels. We are in the process of investigating these tradeoffs further.

Several probe and gauge technologies have been integrated into the event propagation [46] and network layers, often in hardware via SNMP [47]. These tend to be optimized for lower-level, high-volume general-purpose packet streams. They can easily be utilized by KX, which can provide higher-level semantics to simple matches found in these lower layers.

"Grid Computing" attempts to make distributed computing resources visible as a single virtual computer. One of the most extensively developed Grid computing platforms is the Globus toolkit [48]. Grid computing is a natural match for the automated distributed management capabilities of our KX architecture. Features such as system configuration management and autonomic management have been listed as desiderata for commercial Grid computing [49], but are not currently part of the Grid computing standards.

## 6. Conclusions and Future Work

We have discussed KX, an implementation of an easily-integrable external monitoring infrastructure, defined as a component-replaceable meta-architecture, which can be used to add autonomic self-management and self-healing functionality to legacy systems and large-scale systems of systems. We have focused on the Event Packager and Event Distiller, components, which have not been explained in prior publications. Our examples, for failure detection, load balancing, and email processing, demonstrate the success of our solution.

We are investigating extensions of reported ongoing research in several directions. In particular, making KX internals more autonomic is a major goal – automatic probe deployment, automatic gauge derivation, and automated construction of repair plans are under investigation. Our ongoing work with CMU to integrate architectural semantics into the framework should help.

## 7. Acknowledgements

We would like to thank the other members of the Programming Systems Lab for their contributions to this effort, as well as our outside colleagues who alternately provided criticism and encouragement – Bob Balzer, Dave Wile, David Garlan, Bradley Schmerl, David Wells, Nathan Combs, George Heineman, Bob Neches, Lee Osterweil and John Salasin. The Programming Systems Laboratory is funded in part by Defense Advanced Research Project Agency under DARPA Order K503 monitored by Air Force Research Laboratory F30602-00-2-0611, by National Science Foundation grants CCR-0203876, EIA-0071954, and CCR-9970790, and by Microsoft Research and IBM. The software described here can be downloaded for research and education purposes from http://www.psl.cs.columbia.edu.

## 8. References

[1] IBM Research, *Autonomic Computing*, http://www.research.ibm.com/autonomic.

[2] Microsoft, *Windows XP/Office XP Feature Overview*, http://www.microsoft.com/windowsxp/pro/evaluation/overviews/windowsxpofficexp.asp.

[3] Feder, Barnaby J. "On the Trailing Edge of the Arms Industry, by Choice." *The New York Times*, March 30, 2003.

[4] Raijlich, V., Wilde, N., Buckellew, M., and Page, H. "Software cultures and evolution." *IEEE Computer*, Vol. 34, Iss. 9, Sep. 2001, pp. 24-28.

[5] Bekker, S. "Microsoft Error Reporting Drives Bug Efforts", *ENT News*, October 3, 2002, http://www.entmag.com/news/article.asp?EditorialsID=5532.

[6] SANS, "What is Host-Based Intrusion Detection?" *Intrusion Detection FAQ*, http://www.sans.org/resources/idfaq/host_based.php.

[7] LANDesk Software, *LANDesk Management Software*, http://www.landesksoftware.com/.

[8] Valetto, G. and Kaiser, G. "Using Process Technology to Control and Coordinate Software Adaptation." *International Conference on Software Engineering*, May 2003, in press.

[9] Salasin, J. *DARPA DASADA Program*, http://www.rl.af.mil/tech/programs/dasada/program-overview.html.

[10] Carzaniga, A., Rosenblum, D.S. and Wolf, A.L. "Design and Evaluation of a Wide-Area Event Notification Service." *ACM Transactions on Computer Systems*, 19(3):332-383, Aug. 2001.

[11] Gross, P.N, Gupta, S., Kaiser, G.E., Kc, G.S., Parekh, J.J. "An Active Events Model for System Monitoring." *Working Conference on Complex and Dynamic Systems Architectures, December 2001.*

[12] BBN, *Cougaar*, http://www.cougaar.org/.

[13] Valetto, G., Kaiser, G.E. and Kc, G. “A Mobile Agent Approach to Process-based Dynamic Adaptation of Complex Software Systems.” *Eighth European Workshop on Software Process Technology*, LNCS 2077, June 2001, pp. 102-116.

[14] Kaiser, G., Stone, A. and Dossick, S. “A Mobile Agent Approach to Lightweight Process Workflow.” *Position paper in International Process Technology Workshop*, September 1999.

[15] Programming Systems Lab, *Download Page*, http://www.psl.cs.columbia.edu/download.

[16] Segall, B., Arnold, D., Boot, J., et. al. “Content-based routing with Elvin4.” *Proceedings of AUUG2K*, June 2000.

[17] Perrochon, L. *Using Context-Based Correlation in Network Operations Management,* http://pavg.stanford.edu/cep/cidf.ps.gz.

[18] Garlan, D., Monroe, R., and Wile, D. “Acme: An Architecture Description Interchange Language.” *Proceedings of CASCON '97*, November 1997.

[19] CAUCE, *About Spam*, http://www.cauce.org/about/problem.shtml.

[20] Sendmail Inc., *Sendmail Mail Server*, http://www.sendmail.org/.

[21] Sendmail Inc., *Sendmail Mail Filter API*, http://www.sendmail.com/partner/resources/development/milter_api/.

[22] Programming Systems Lab, *Event Distiller Documentation*, http://www.psl.cs.columbia.edu/xues/EventDistiller.html.

[23] SpamAssasin, *Spam Filter*, http://www.spamassassin.org.

[24] ISI, *GeoWorlds GIS System*, http://www.isi.edu/geoworlds/.

[25] Sun, *Jini Technology*, http://www.sun.com/software/jini/.

[26] Heineman, G., Calnan, P., Kurtz, B., et. al. *Active Interface Development Environment (AIDE)*. http://www.cs.wpi.edu/~heineman/dasada/.

[27] ACME, *AcmeStudio Development Environment*, http://www-2.cs.cmu.edu/~acme/AcmeStudio/AcmeStudio.html.

[28] Telecom Italia (TILAB), http://www.telecomitalialab.com/.

[29] OBJS, *Software Surveyor*, http://www.objs.com/DASADA/.

[30] Garlan, D., Schmerl, B., and Cheng, J. “Using Gauges for Architecture-Based Monitoring and Adaptation.” *Working Conference on Complex and Dynamic Systems Architectures,* December 2001.

[31] Honeywell, *Honeywell DASADA Project*, http://www.htc.honeywell.com/projects/DASADA/.

[32] Cobleigh, J., Osterweil, L., Wise, A., and Lerner, B. “Containment Units: A Hierarchically Composable Architecture for Adaptive Systems.” *Proceedings of the 10th International Symposium on the Foundations of Software Engineering (FSE 10)*, pp. 159-165.

[33] Wolf, A., Heimbigner, D., Knight, J.C., Devanbu, P.T., Gertz, M., Carzaniga, A. “Bend, Don't Break: Using Reconfiguration to Achieve Survivability.” *Third Information Survivability Workshop--ISW-2000*, October 2000.

[34] van Renesse, R. and Binnan, K.P., "Astrolabe: A Robust and Scalable Technology for Distributed System Monitoring, Management, and Data Mining." *ACM TOCS*, November 2001.

[35] Konstantinou, A.V., Yemini, Y., and Florissi, D. “Towards Self-Configuring Networks.” *DARPA Active Networks Conference and Exposition (DANCE)*, May 2002.

[36] OC Systems, “Aprobe: A New Approach for Testing Web Applications.” http://www.ocsystems.com/aprobe_web_testing.html.

[37] OC Systems, “Improving Availability of Enterprise Applications with RootCause.” http://www.ocsystems.com/rootcause_white_paper.html.

[38] System Management ARTS, http://www.smarts.com.

[39] Sterritt, R., Shapcott, C.M., Adamson, K., and Curran, E.P. "High Speed Network First-Stage Alarm Correlator." *International Conference Intelligent Systems And Control*, 2000, pp 391-397.

[40] Steinder, M. and Sethi, A.S. “Probabilistic event-driven fault diagnosis through incremental hypothesis updating.” *IFIP/IEEE Symposium on Integrated Network Management*, 2003.

[41] Internet Security Systems, *RealSecure Network Protection*, http://www.iss.net/products_services/enterprise_protection/rsnetwork/.

[42] Cisco, *Cisco Intrusion Detection System*, http://www.cisco.com/univercd/cc/td/doc/pcat/nerg.htm.

[43] Lee, W., Stolfo, S.J. and Chan, P.K. “Learning Patterns from Unix Process Execution Traces for Intrusion Detection”, *Proc. AAAI-97 Work. on AI Methods in Fraud and Risk Management*, 1997.

[44] Luckham, D.C.; Vera, J. “An event-based architecture definition language.” *IEEE Transactions on Software Engineering*, Vol. 21, Iss. 9, Sep. 1995, pp. 717-734.

[45] Yemini, S.A., Kliger, S., et. al. “High speed and robust event correlation.” *IEEE Communications Magazine*, Vol. 34 Issue 5, May 1996, pp. 82-90.

[46] Zhao, Y. and Strom, R. “Exploiting Event Stream Interpretation in Publish-Subscribe Systems.” *Principles of Distributed Computing*, 2001.

[47] Rose, M., ed. *RFC 1052: A Convention for Defining Traps for use with the SNMP*, 1991, http://www.ietf.org/rfc/rfc1215.txt.

[48] Foster, I., Kesselman, C., et. al. “The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration.” *Open Grid Service Infrastructure WG, Global Grid Forum*, June 22, 2002.

[49] Kishimoto, H., Snelling, D. “OGSA Fundamental Services: Requirements for Commercial GRID Systems.” *Global Grid Forum Draft*, October 3, 2002.

## A1. Event Packager and Event Distiller ruleset examples

While example ED rule snippets were provided in the body of the paper, we give more comprehensive rulesets here and describe their behavior.

### A1.1. Event Packager Rulebase

We give an example here of a configuration that subscribes to one Siena event bus for events which have the attribute-value pair ("TestAttribute", "TestValue"), stores the results in a SQL database, and finally copies the results to another Siena event bus.

```
<EventPackagerConfiguration>
 <Inputters>
  <Inputter Name="SienaInput1" Type="psl.xues.ep.input.SienaInput"
            SienaReceivePort="7890">
   <SienaFilter Name="TestFilter1">
    <SienaConstraint AttributeName="TestAttribute" Op="="
            ValueType="String" Value="TestValue" />
   </SienaFilter>
  </Inputter>
  <Inputter Name="ConsoleInput1"
            Type="psl.xues.ep.input.ConsoleInput" />
 </Inputters>
 <Outputters>
  <Outputter Name="SienaOutput1"
             Type="psl.xues.ep.output.SienaOutput"
             SienaReceivePort="7891" />
  <Outputter Name="NullOutput1" Type="psl.xues.ep.output.NullOutput" />
 </Outputters>
 <Transforms>
  <Transform Name="Store1" Type="psl.xues.ep.transform.StoreTransform"
             StoreName="HSQLDB1" />
 </Transforms>
 <Stores>
  <Store Name="HSQLDB1" Type="psl.xues.ep.store.JDBCStore"
            DBType="hsqldb" DBDriver="org.hsqldb.jdbcDriver"
            DBName="xues" DBTable="xues" Username="sa"
            Password="" />
 </Stores>
 <Rules>
  <Rule Name="TestRule1">
   <Inputs><Input Name="SienaInput1" /></Inputs>
   <Transforms><Transform Name="Store1" /></Transforms>
   <Outputs><Output Name="SienaOutput1" /></Outputs>
  </Rule>
  <Rule Name="ConsoleRule">
   <Inputs><Input Name="ConsoleInput1" /></Inputs>
   <Outputs><Output Name="NullOutput1" /></Outputs>
  </Rule>
 </Rules>
</EventPackagerConfiguration>
```

The references to the Console are needed if console-level control is desired of the Event Packager; it perceives the user console to be yet another source (and, potentially, a sink) for events.

### A1.2. Event Distiller Rulebase

We provide here an extended view of the rules presented in Figure 6, including the corresponding notifications if the pattern is matched.

```
<rulebase xmlns="http://www.psl.cs.columbia.edu/2001/01/DistillerRule.xsd">

<rule name="ActiveEvent">
 <states>
  <state name="Start" timebound="-1" children="End" actions=""
         fail_actions="">
   <attribute name="Service" value="*service"/>
   <attribute name="Status" value="Started"/>
   <attribute name="ipAddr" value="*ipaddr"/>
   <attribute name="ipPort" value="*ipport"/>
   <attribute name="time" value="*time"/>
  </state>
  <state name="End" timebound="15000" children="" actions="Debug"
         fail_actions="Crash">
   <attribute name="Service" value="*service"/>
   <attribute name="State" value="FINISHED_STATE"/>
   <attribute name="ipAddr" value="*ipaddr"/>
   <attribute name="ipPort" value="*ipport"/>
   <attribute name="time" value="*time2"/>
  </state>
 </states>
 <actions>
  <notification name="Crash">
   <attribute name="Notification_Type" value="GW_Alarm"/>
   <attribute name="Message" value="Dead_Service"/>
   <attribute name="KX_Reaction_Type" value="Workflow"/>
   <attribute name="KX_Reaction_Spec" value="Disable_Service"/>
   <attribute name="Timestamp" value="*time"/>
   <attribute name="Service" value="*service"/>
   <attribute name="Name" value="gwHostAdapter"/>
   <attribute name="IPaddress" value="*ipaddr"/>
   <attribute name="port" value="*ipport"/>
   <attribute name="serviceURI" value="http://www.isi.edu/..."/>
   <attribute name="schemaURI" value="http://www.isi.edu/..."/>
  </notification>
  <notification name="Debug">
   <attribute name="GWFinish" value="Yes"/>
   <attribute name="Timestamp" value="*time2"/>
  </notification>
 </actions>
</rule>

</rulebase>
```

In this case, we allocated 15 seconds for the method to complete, and in the Crash case, both static and dynamic (i.e., wildcard-bound) data were reported (certain URLs were commented out to keep the length to a minimum).

Details on individual keywords in either rulebase can be found on the PSL website (see http://www.psl.cs.columbia.edu/xues).